TyTAN: Tiny Trust Anchor for Tiny Devices

Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann
Technische Universität Darmstadt (EC-SPRIDE), Germany

Patrick Koeberl
Intel Labs

ABSTRACT

Embedded systems are at the core of many security-sensitive and safety-critical applications, including automotive, industrial control systems, and critical infrastructures. Existing protection mechanisms against (software-based) malware are inflexible, too complex, expensive, or do not meet real-time requirements.

We present TyTAN, which, to the best of our knowledge, is the first security architecture for embedded systems that provides (1) hardware-assisted strong isolation of dynamically configurable tasks and (2) real-time guarantees. We implemented TyTAN on the Intel® Siskiyou Peak embedded platform and demonstrate its efficiency and effectiveness through extensive evaluation.

1. INTRODUCTION

Today, millions of embedded systems are used in safety and security critical applications. Current industrial trends and initiatives aim to "connect the unconnected" to realize the Internet of Everything, where embedded systems[1] play the central role. These systems generate, process, and exchange vast amount of security and safety critical data as well as privacy sensitive information, and hence are appealing targets of various attacks. Recent studies have revealed many security vulnerabilities in embedded devices [3, 4, 16, 19, 11, 2, 15, 23, 8]. This poses new challenges on the design and implementation of secure embedded systems that typically must provide multiple functions, basic security features, and real-time guarantees at minimal cost. To ensure the correct operation of these devices, it is crucial to assure their integrity, in particular of their code and data.

While most hardware security solutions, such as Trusted Platform Modules (TPMs) [27], do not scale to embedded systems because of their high complexity and costs [28, 26, 18, 14], software-based solutions [9, 22, 21, 12] typically rely on strong assumptions that are hard to achieve in practice [1]. On the other hand, approaches that target low-end embedded devices do not meet the real-time requirements of many embedded applications, or are highly inflexible, for instance, they assume a static software configuration and do not allow dynamic loading of applications at runtime [6, 25, 17, 10]. We elaborate on these proposals later in the related work section.

Contribution. Our contributions are as follows:

A security architecture for tiny devices. We present TyTAN, which, to the best of our knowledge, is the first security architecture for low-end embedded systems that provides (1) a hardware-assisted dynamic root of trust, allowing secure task loading at runtime; (2) secure inter-process communication (IPC); (3) local and remote attestation; and (4) real-time guarantees. TyTAN is designed for multi-stakeholder scenarios and allows for secure execution of mutually distrusting tasks.

Implementation. We implemented TyTAN on Intel® Siskiyou Peak [20], an architecture intended for deeply embedded systems.

Evaluation. We evaluated TyTAN's efficiency and effectiveness. We show that all of TyTAN's components are real-time compliant and demonstrate its applicability to automotive embedded control systems.

Outline. We introduce the model of TyTAN in Section 2. We describe TyTAN's architecture in Section 3, and our implementation in Section 4. We discuss TyTAN's security in Section 5, and provide our evaluation results in Section 6. Eventually, we discuss related work in Section 7 and conclude in Section 8.

2. MODEL AND REQUIREMENTS

The model involves the following parties: the *device manufacturer* $\mathcal{M}$, the *device owner* $\mathcal{O}$, and multiple *task providers* $\mathcal{P}_i$. On embedded systems applications are usually called *tasks*. $\mathcal{M}$ provides the underlying platform hardware and the software components which are critical for the correct operation of TyTAN (marked as *trusted software* in Figure 1). Hence, these parts must be trusted by all parties. $\mathcal{O}$ controls the operating system (OS). The OS and the tasks provided by $\mathcal{P}$ (e.g., Task A - D in Figure 1) are mutually untrusted. All tasks are isolated from each other and *secure tasks* are in addition isolated from the OS.

We focus on resource-constrained embedded systems as

[1]The term *embedded system* is widely used for a large variety of systems ranging from microcontrollers with minimal functionality to quite powerful systems such as smartphones and enterprise routers [4]. In this paper, we focus on resource-constrained embedded systems.

DAC '15, June 07 - 11, 2015, San Francisco, CA, USA

http://dx.doi.org/10.1145/2744769.2744922.

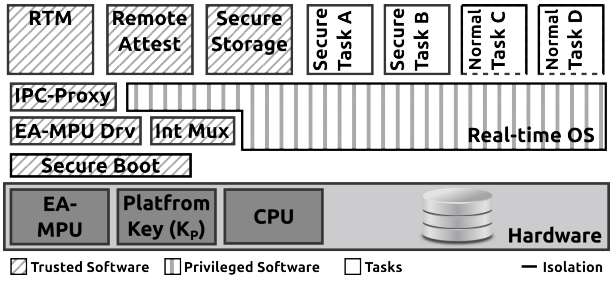

Figure 1: TyTAN system architecture

used in many automotive and industrial applications. Examples are the TI MSP430, ARM Cortex M3, and Intel® Siskiyou Peak which we use as base platform for TyTAN.

Safety and security critical applications of embedded systems in, for instance, automotive, require: (1) real-time guarantees; (2) isolation of system components; (3) dynamic configuration; (4) techniques for device integrity verification (attestation); and (5) support for multiple, potentially untrusting, stakeholders.

Real-time guarantees. Acting reliably within strict time frames is highly relevant but not considered by most security architectures for embedded devices [6, 25, 17].

Isolation. Faults in one system component cannot (directly) influence other components. Isolation is fundamental to protect critical components against unintended access by other (malicious) components.

Dynamic configuration. Tasks (applications) can be dynamically loaded, unloaded, started, and stopped on demand at runtime, increasing efficiency through better resource utilization. Tasks can be updated, which is necessary to address software flaws.

Attestation. While *local attestation* allows different components on the same system to mutually verify their integrity, *remote attestation* allows a device to prove the integrity of its software state to another device.

Multiple stakeholders. Today, embedded devices execute tasks from multiple, mutually distrusting stakeholders. For instance, automotive electronic control units often run software provided by the component supplier and the car manufacturer. While the component supplier requires protecting its intellectual property and the integrity of its software components, the car manufacturer wants to ensure the correct and reliable operation of its tasks.

3. ARCHITECTURE

TyTAN, see Figure 1, is composed of several components, as described in the following.

Tasks. TyTAN supports two types of tasks: *normal tasks* are isolated from other tasks but accessible to the OS; and *secure tasks* that are isolated from all other software including the OS. Each task t has a unique identifier id_t, i.e., the hash digest of its binary code. Tasks are loadable, unloadable, and suspendable at runtime.

EA-MPU. TyTAN is based on an *Execution-aware Memory Protection Unit* (EA-MPU) [10], a hardware component providing: (1) memory access control enforcement based on the code that aims to access a data region, e.g., the stack of a task can be accessed only by the task itself but no other software; (2) each task can be invoked only at a dedicated entry point; and (3) interrupts are handled such that the memory access control rules of the EA-MPU are enforced, i.e., a (malicious) interrupt handler cannot gain any information about the state of an interrupted task.

Platform Key. The TyTAN hardware platform comes with a *platform key* K_p. Access to this key is controlled by the EA-MPU and only trusted software components have access to it. Additional keys can be derivated from K_p, e.g., for remote attestation or for secure storage.

Real-time OS. TyTAN provides real-time scheduling. It ensures that all tasks and system components can be interrupted to allow other pending operations to proceed within the time frame allocated to them.

Secure boot. TyTAN's trusted software components (i.e., EA-MPU driver, Int Mux, IPC Proxy, RTM task, Remote Attest and Secure Storage) are loaded with secure boot and isolated from the rest of the system by the EA-MPU to ensure their integrity. These components are not higher privileged than the OS and only some of them have the same privileges as the OS (e.g., EA-MPU driver).

Trusted execution. TyTAN provides trusted execution by isolating secure tasks and trusted software components based on the access control enforced by the EA-MPU. Each of the security primitives described in the following is isolated from all other system components.

EA-MPU driver. The dynamic handling of tasks requires the EA-MPU to be dynamically configurable. This is performed by the EA-MPU driver, which sets the memory access control rules in the EA-MPU when loading or unloading a secure task. The EA-MPU rules for the static components (including the EA-MPU driver itself) are set during secure boot.

Attestation. To prove the integrity of a task t to a local or remote verifier, the *Root of Trust for Measurement* (RTM) task computes a cryptographic hash function over the binary code of each created task. This hash digest serves as identity of the task id_t. To meet real-time requirements, the RTM task must be interruptible during the hash calculation. By isolating t's memory and preventing its execution, TyTAN ensures that t is immutable while the RTM task computes id_t. This guarantees the reliable verification of id_t.

The authenticity of id_t and its origin is crucial. TyTAN supports different authentication methods for local and remote attestation. The EA-MPU ensures that only the RTM task can modify id_t. For local attestation, id_t can be used as both identifier and attestation report of t. Remote attestation on TyTAN uses Message Authentication Codes (MAC) along with an attestation key K_a to prove the authenticity of id_t to a remote verifier. K_a is derivated from K_p and only accessible to the *Remote Attest* task.[2]

Secure inter-process communication (IPC). TyTAN enables secure communication between tasks via an IPC proxy, which forwards the message m from the sender $\mathcal{S}$ to the receiver $\mathcal{R}$. $\mathcal{S}$ copies m and the identity $id_{\mathcal{R}}$ of $\mathcal{R}$ to the CPU registers and invokes the IPC proxy by an software interrupt.[3] The IPC proxy determines $\mathcal{R}$'s memory location and writes m and $id_{\mathcal{S}}$ to $\mathcal{R}$'s memory. This implicitly authenticates m and $id_{\mathcal{S}}$ since the EA-MPU ensures

[2]In [17] a key derivation scheme is shown which allows the creation of individual attestation keys per $\mathcal{P}$.

[3]Provisioning $\mathcal{S}$ with $id_{\mathcal{R}}$ is left to the task developer.

that only the IPC proxy can write to $\mathcal{R}$'s memory. To efficiently transfer large amount of data between tasks, the IPC proxy sets up shared memory that is accessible only to the communicating tasks.

Secure storage. Secure storage is realized as a secure task. For each task a *task key* $K_t = HMAC(id_t|K_p)$ is generate which is bound to the task identity (id_t) and the platform (K_p). Tasks interact with the secure storage task over secure IPC which allows the identification of the requesting task. All data a task sends to the secure storage task get encrypted with K_t. Since id_t is included in K_t a task that tries to access data stored before will only succeed if it has the same id_t as the task that stored the data, i.e., if it is the same tasks.

4. IMPLEMENTATION

Hardware platform.

We implemented TyTAN on the Intel® Siskiyou Peak architecture [20], a low-power, 32-bit core intended for embedded applications. Siskiyou Peak uses a flat, physical addressing model and interacts with peripherals using memory-mapped input/output (MMIO).

We extend the static EA-MPU usages presented in TrustLite [10] with dynamic configuration of memory access control rules. We implemented TyTAN on a Xilinx Spartan-6 FPGA running at 48 MHz.

Operating System.

TyTAN uses the FreeRTOS[4] real-time operating system. We ported FreeRTOS to Siskiyou Peak and extended FreeRTOS with dynamic handling of secure tasks and support for the EA-MPU. Our extensions to FreeRTOS do not violate real-time requirements: (1) multi-tasking support, (2) priority-based pre-emptive scheduling, (3) bounded execution time for primitives, (4) high-resolution real-time clock, (5) special alarms and time-outs, (6) real-time queuing, and (7) delaying of processes (interrupt/resume task execution) [24]. Specifically, we extended FreeRTOS's preemptive scheduler to support secure tasks and designed all software components of TyTAN to be interruptible, or to have an upper bound on their execution time.

Dynamic task handling. Loading tasks at runtime requires: (1) allocation of memory for the new task; (2) loading the task into memory and preparing its stack: FreeRTOS operates on physical memory and the base address of a task changes depending on which memory regions are free at load time, making relocation necessary; and (3) invocation of the task, i.e., adding it to the OS scheduler.

To perform those steps we extended FreeRTOS with an ELF[5] loader. ELF supports relocatable binaries and encodes all information required for relocation in ELF file headers.

Unloading a task requires deleting it from the OS scheduler and reclaiming its memory. Task suspending requires the OS scheduler to maintain a list of tasks that are loaded but should not be executed at the moment.

Secure tasks. Secure tasks are isolated from other software components, i.e., the memory of a secure task can be accessed only by the task itself and trusted system components. The OS is not trusted and cannot access the task's memory and the EA-MPU enforces that secure tasks are invoked only at a dedicated entry point to prevent code reuse attacks.[6] We adapted FreeRTOS to consider these restrictions when interrupting and (re)starting secure tasks.

Interrupting secure tasks. Tasks are frequently interrupted, e.g., to react to an event like an arriving network package. Whenever an interrupt occurs, the hardware exception engine stops the current task and executes a predefined routine, called interrupt hander, which reacts to the interrupt and resumes the task afterwards. After the interrupt, the interrupted task should continue execution as if it had never been interrupted,[7] which requires the interrupt handler not to change the state of the task. The task's state consists of the content of the task's memory and the CPU registers (known as the *context* of the task). While the task's memory typically remains unchanged, the CPU registers are used by the interrupt handler and must be saved. The *instruction pointer* (EIP) and *flags register* (EFLAGS) are saved by the exception engine to the stack of the interrupted task. For normal tasks, all other CPU registers are saved by the interrupt handler to the task's stack. Since the context and stack of a secure task may contain sensitive information, the untrusted OS may not be able to access this data. TyTAN uses the trusted interrupt multiplexer (*Int Mux*) to securely save the context of a task to it's stack before control is passed to the interrupt handler. Alternatively, saving the task's context to its stack can be implemented in hardware, reducing latency at the cost of additional hardware.

(Re)starting secure tasks. When a normal task is resumed after it has been interrupted, its context is loaded from the task's stack to the CPU registers. The registers saved by the hardware (EIP and EFLAGS) are restored by a dedicated CPU instruction, which also continues to execute the task from the point where it has been interrupted. When a new task is created, the OS prepares the stack of this task as if it had been executed before and was interrupted. Then the OS resumes the task and loads the initial context of the task to the CPU registers.

For secure tasks there are two restrictions: (1) the OS cannot access the stack of a secure task and restore its context; and (2) secure tasks can be invoked only with a dedicated entry routine. This entry routine detects whether the task has been (re)started or was invoked to receive a message and acts accordingly. TyTAN provides this information in a CPU register, which is checked by the entry routine. When the task has been (re)started, the entry routine restores the task's context and continues the task's execution. Since the entry routine is similar for all secure tasks, it is automatically included by the TyTAN tool chain and does not need to be implemented by the task programmer.

RTM task. When a task t is loaded, the RTM task computes the hash digest of the code, static data, and initial stack layout of t.[8] This measurement is the basis for local and remote attestation on TyTAN. The integrity of the RTM task is protected by secure boot and the EA-MPU. As described before, during the loading of t it is subject to relocation. A measurement of a "relocated" task would only

[4]RTOS available under GPL (`http://www.freertos.org`)
[5]Executable Linking Format

[6]Code reuse attacks pose a severe threat on diverse platforms including embedded systems [7].
[7]If the task requests a service from the OS via an interrupt this has of course an effect on the task but this is intended.
[8]We use SHA-1 but other hash algorithms can also be used.

be verifiable with additional information, e.g., the memory location at which the task is loaded. To provide a position-independent measurement for tasks, the RTM task temporarily reverts the changes made during relocation before computing the hash digest.

Loading tasks. A new task t is loaded as follows: (1) the OS allocates memory for t; (2) loads t into memory performing relocation; (3) prepares the stack; then (4) the EA-MPU is configured to protect the memory of t; (5) t is measured; and (6) the OS is notified to schedule t.

Secure IPC. The sender $\mathcal{S}$ loads the message m and the identity $id_{\mathcal{R}}$ of the receiver $\mathcal{R}$ (i.e., the measurement[9] of $\mathcal{R}$) into the CPU registers and issues an interrupt. This invokes the IPC proxy, which obtains the origin of the interrupt from the hardware and determines $\mathcal{S}$'s identity $id_{\mathcal{S}}$. The memory location of $\mathcal{R}$ is stored by the RTM task, which maintains a list of the identities of all loaded tasks and their memory addresses. Then the IPC proxy writes m and $id_{\mathcal{S}}$ to the memory of $\mathcal{R}$. For synchronous communication, the IPC proxy branches to $\mathcal{R}$, whose entry routine processes m. For asynchronous communication, the IPC proxy continues executing $\mathcal{S}$ and $\mathcal{R}$ processes m the next time it is scheduled.

Interrupts. Interrupts are handled by different interrupt handlers, which are determined by the *interrupt descriptor table* (IDT). To ensure the correct use of interrupt handlers, the integrity of the IDT is protected by the EA-MPU. The register pointing to the IDT is static and cannot be modified to install another (malicious) IDT.

5. SECURITY CONSIDERATIONS

The primary goal of TyTAN is to assure the integrity of critical software components and secure tasks. This is achieved through secure boot and hardware-enforced memory access control.

Another important property of TyTAN is real-time execution of tasks, which relies on the availability of the platform. There are different attack vectors that an external adversary can leverage to undermine the availability of TyTAN: Denial of Service (DoS), and compromising the platform's software in order to disturb the operation of the system.

Denial of service attacks are domain specific (e.g., network flooding if a network interface exists, or disconnection the power supply if the device is physically accessible), and no general solution exists to prevent DoS attacks.

To disturb the operations of the system the adversary needs to gain control over the OS or a trusted software components, e.g., the EA-MPU driver. Normal tasks as well as secure task cannot disturb the operations of other components of TyTAN, due to the fact that they are isolated, and bound in their use of system recourse (e.g., execution time or memory). Hence, the adversary who controls a task[10] cannot disturb the availability of the platform. Only if the adversary can exploit a vulnerability in the OS to gain higher privileges he can succeed in his attack.

6. EVALUATION

We evaluate the performance and applicability of TyTAN for embedded control systems in automotive environments.

[9]For enhanced performance, our implementation uses only the first 64 bits of the hash digest.

[10]The attacker might be a task provider ($\mathcal{P}$) how deployed a malicious task, or the attacker compromised a benign task

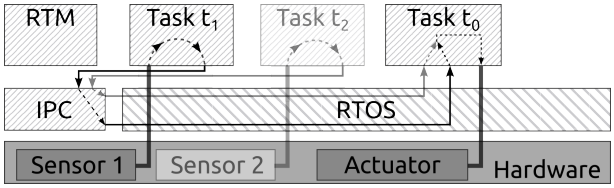

Figure 2: Use-case scenario

Table 1: Use-case evaluation results

Task	t_1	t_2	t_0
Before loading t_2	1.5 kHz	—	1.5 kHz
While loading t_2	1.5 kHz	—	1.5 kHz
After loading t_2	1.5 kHz	1.5 kHz	1.5 kHz

To validate TyTAN's real-time properties, we evaluate the performance of each of its components.

Use-case Evaluation.

Our use case concerns a simulated adaptive cruise control system, where an embedded device controls the speed of a vehicle depending on the accelerator pedal position and the speed of a vehicle in front (measured by a radar sensor). The device runs three secure tasks (see Figure 2). Task t_1 permanently monitors the accelerator pedal position sensor. Task t_2 is loaded on demand when adaptive cruise control is activated by the driver and then monitors the radar sensor. Task t_0 controls the speed of the vehicle based on the data provided by t_1 and t_2 (t_0 implements the engine control software). When cruise control is activated, t_2 is loaded into memory, which involves relocation, preparing the stack, and measuring t_2. All these operations take 27.8 ms, which is longer than the time available between two scheduling cycles of t_0 and t_1. Hence, loading t_2 could block t_0 and t_1 if the loading procedure was not interruptible. Our results in Table 1 show that despite frequently scheduling of t_0 and t_1 they still meet their deadlines while t_2 is loaded, which is crucial for safe and precise control of the vehicle's speed.

Performance of TyTAN Components.

We evaluated the performance of TyTAN, including all components that could have an impact on its real-time behavior, namely: (1) interrupt handling; (2) secure task creation; and (3) secure IPC.

We present all results in *clock cycles* since the clock-speed of a platform is variable and depends on many factors that are not related to TyTAN.

Interrupt handling. The interrupt handler: (1) saves the context, (2) wipes the CPU registers, and (3) branches to the routine handling the interrupt. Table 2 compares the results with the unmodified FreeRTOS. Continuing the execution of an interrupted task requires branching to this task and restoring its context. Table 3 shows the evaluation results for restoring a secure task and compares them to FreeRTOS.

Creating tasks. Creating a secure task t requires: relocating t; configuring the EA-MPU for t; and measuring t. Table 4 shows the performance results for creating task.[11]

Relocation. The performance of relocation depends on the number n of addresses changed in task t by the relocation process. Table 5 shows the results for different n, which

[11]With 9 relocations and a memory size of 3,962 Bytes.

Table 2: Performance of saving the context of a secure task (in clock cycles)

Store context	Wipe registers	Branch	Overall	Overhead
38	16	41	95	57

Table 3: Performance of restoring the context of a secure task (in clock cycles)

Branch	Restore	Overall	Overhead
106	254	384	130

indicate that the runtime of relocation is linear in n.

EA-MPU configuration. Configuring the EA-MPU requires: finding a free EA-MPU slot for the new access control rule; checking the new rule against existing EA-MPU rules (i.e., that protected regions do not overlap); and writing the rule to the EA-MPU (see Table 6).

Task measurement. The time required to measure a task t depends on: the memory size of t; the number of memory addresses in t changed by relocation; and the number of interruptions of the RTM task during measuring t. Table 7 shows the performance results for measuring a task. It shows that the runtime (T) of measuring a task depends on the number of blocks (b) and the number of addresses (a) to handle: $T \approx 4300\,clock\,cycles + (b \cdot 3900\,clock\,cycles) + 100\,clock\,cycles + (a \cdot 500\,clock\,cycles)$.

The measurement is not required for *normal tasks*.

Secure IPC. The communication performance depends on the runtime of: the IPC proxy ($1,208\,clock\,cycles$); and the entry routine of the receiver processing the message ($116\,clock\,cycles$). Hence, the overall performance of the sceure IPC mechanism is $1,324\,clock\,cycles$

Memory consumption.

The memory consumption of TyTAN's OS is the amount of memory used when no task is loaded. Table 8 compares the memory consumption of TyTAN and FreeRTOS.

Secure tasks implement an entry routine to handle interrupts, which slightly increases the memory consumption of secure tasks compared to normal tasks.

7. RELATED WORK

There is a rich body of literature on security architectures for embedded systems, mainly due to the broad range of devices considered as embedded systems [4, 3]. On the upper end are the Intel® and ARM® architectures, which are widely used in mobile devices (e.g., smartphones and tablets). For these systems, a variety of security architectures have been proposed: software-based isolation and virtualization [13]; trusted computing based on secure hardware (e.g., Trusted Platform Module (TPM) [27]); and processor architectures providing secure execution [28, 26, 18, 14]. However, all these approaches are too complex and expensive for low-end embedded systems. Security solutions for such devices are typically based on hardware-enforced isolation of security-critical code and data from other software on the same platform. The most prominent examples include, SMART [6], SPM [25], SANCUS [17], and TrustLite [10]. SMART protects the integrity of only one specific task with read-only memory, which does not allow code changes after deployment. The integrity protected task may not be interrupted rendering SMART incompatible for real-time systems. SPM provides hardware-enforced isolation of tasks by granting access to a task's data region only to the task itself. However, these tasks have a fixed memory layout and cannot be interrupted. Further, the task measurement of SPM is performed in hardware, i.e., it is non-interruptible and at the same time dependent on the memory size of the measured task, which violates real-time system requirements. SANCUS extends SPM with a mechanism to generate and manage cryptographic secrets of tasks but inherits SPM's limitations, e.g., no secure interrupts. The secure interrupt mechanism introduced for Sancus aims at making the platform suitable for real-time systems [5]. But this mechanism does not fulfil all requirements for a real-time system as identified in [24], e.g., bounded execution time for primitives.

Table 4: Performance of creating a secure task (in clock cycles)

Task type	Relocation	EA-MPU	RTM	Overall	Overhead
Secure	3,692	225	433,433	642,241	437,380
Normal	3,692	225	0	208,808	3,917

Table 5: Performance of relocation for different numbers of addresses changed by relocation (in clock cycles)

# of addresses	Runtime (min)	Runtime (avg)
0	37	37
1	673	703
2	1,346	1,372
4	2,634	2,711

TrustLite generalizes the concept of SPM [25] and SMART [6] and supports interrupting tasks. However, TrustLite requires all software components to be loaded and their isolation to be configured at boot time. In contrast to these works TyTAN provides higher flexibility by providing dynamic loading and unloading of multiple tasks at runtime, secure IPC with sender and receiver authentication, and real-time scheduling.

8. CONCLUSION

We presented TyTAN, the first comprehensive security architecture for low-end embedded systems that provides (1) dynamic loading and configuration of *secure* tasks, (2) secure IPC, and (3) real-time guarantees. We implemented TyTAN on the Intel® Siskiyou Peak architecture and demonstrated its effectiveness and efficiency through extensive evaluation.

Future work includes extending TyTAN with a mechanism to update tasks at runtime (i.e., without stopping and restarting them) to meet the high availability requirements of embedded applications, and new hardware-assisted runtime attacks detection.

Acknowledgement

The authors thank the anonymous reviewers. This work has been co-funded by the German Science Foundation as part of

Table 6: Performance of configuring EA-MPU depending on the position of the first free slot in the EA-MPU with 18 slots in total (in clock cycles)

Free slot position	Finding free slot	Policy check	Writing rule	Overall
1	76	824	225	1,125
2	95	824	225	1,144
18	399	824	225	1,448

Table 7: Performance of measuring a task depending on its memory size and number of memory addresses changed by relocation (in clock cycles)

Memory size	Runtime	# of addresses	Runtime
1 block	8,261	0	114
2 blocks	12,200	1	680
4 blocks	20,078	2	1,188
8 blocks	35,790	4	2,187

project S2 within the CRC 1119 CROSSING, EC-SPRIDE, and the Intel CRI for Secure Computing.

9. REFERENCES

[1] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann. A security framework for the analysis and design of software attestation. In *ACM Conference on Computer & Communications Security (CCS)*. ACM, 2013.

[2] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*. USENIX Association, 2011.

[3] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*. USENIX Association, 2014.

[4] A. Cui and S. J. Stolfo. A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan. In *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2010.

[5] R. de Clercq, F. Piessens, D. Schellekens, and I. Verbauwhede. Secure interrupts on low-end microcontrollers. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, 2014.

[6] K. Eldefrawy, A. Francillon, D. Perito, and G. Tsudik. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[7] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*. ACM, 2008.

[8] A. G. Illera and J. V. Vidal. Lights off! The darkness of the smart meters. In *BlackHat Europe*, 2014.

[9] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *USENIX Security Symposium*. USENIX Association, 2003.

[10] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *European Conference on Computer Systems (EuroSys)*. ACM, 2014.

[11] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy*. IEEE, 2010.

[12] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *Conference on Computer and Communications Security (CCS)*. ACM, 2011.

[13] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE Symposium on Security and Privacy*. IEEE, 2010.

[14] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, 2013.

[15] C. Miller and C. Valasek. A survey of remote automotive attack surfaces. In *BlackHat USA*, 2014.

[16] D. M. Nicol. Hacking the lights out. *Scientific American*, 305, 2011.

[17] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security Symposium*. USENIX Association, 2013.

[18] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan. OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms. In *ACM Conference on Computer & Communications Security (CCS)*. ACM, 2013.

[19] J. Pollet and J. Cummins. Electricity for free — The dirty underbelly of SCADA and smart meters. In *BlackHat USA*, 2010.

[20] J. Rattner. *Extreme scale computing*. ISCA Keynote, 2012.

[21] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2005.

[22] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*. IEEE, 2004.

[23] A. Soullie. Industrial control systems: Pentesting PLCs 101. In *BlackHat Europe*, 2014.

[24] J. A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Systems*, 28(2-3), 2004.

[25] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks*. Springer, 2010.

[26] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *International Conference on Supercomputing (ICS)*. ACM, 2003.

[27] Trusted Computing Group (TCG). Website. `http://www.trustedcomputinggroup.org`, 2011.

[28] J. Winter. Trusted computing building blocks for embedded Linux-based ARM TrustZone platforms. In *ACM Workshop on Scalable Trusted Computing (STC)*. ACM, 2008.

Table 8: Memory consumption of TyTAN's OS

FreeRTOS	TyTAN	Overhead
215,617 Bytes	249,943 Bytes	15.92 %